Lecture 4

Basic searching algorithms

Searching a list

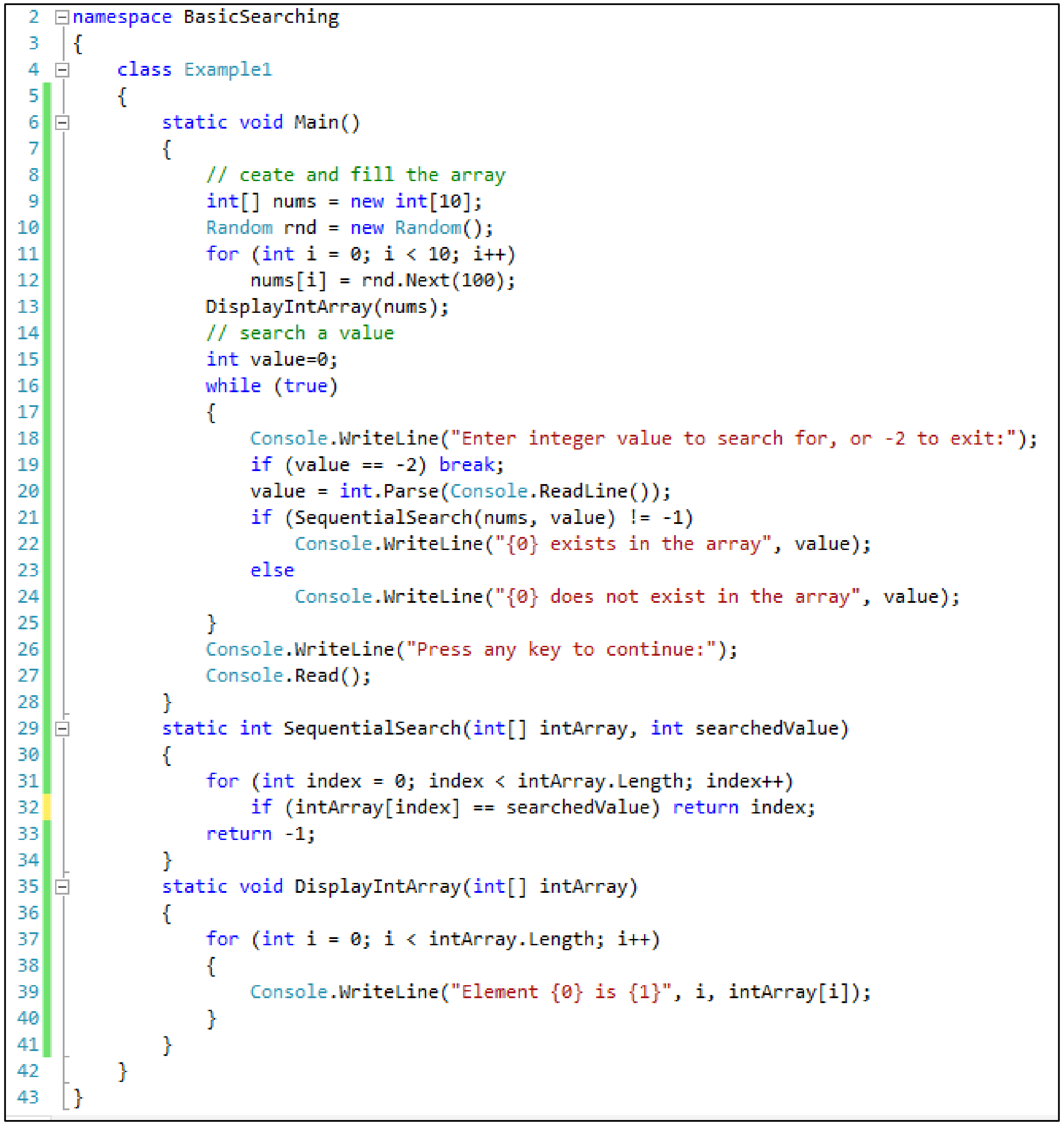
- Sequential search
  - Used with lists in which the items are in random order
- Binary search
  - Used with lists in which the items are sorted

Sequential search

- Begin at the beginning of a set of records (items) and move through each record until you find the record you are looking for or you come to the end of the records
- Is also called a linear search
- Simple to understand and to implement

Sequential search implementation

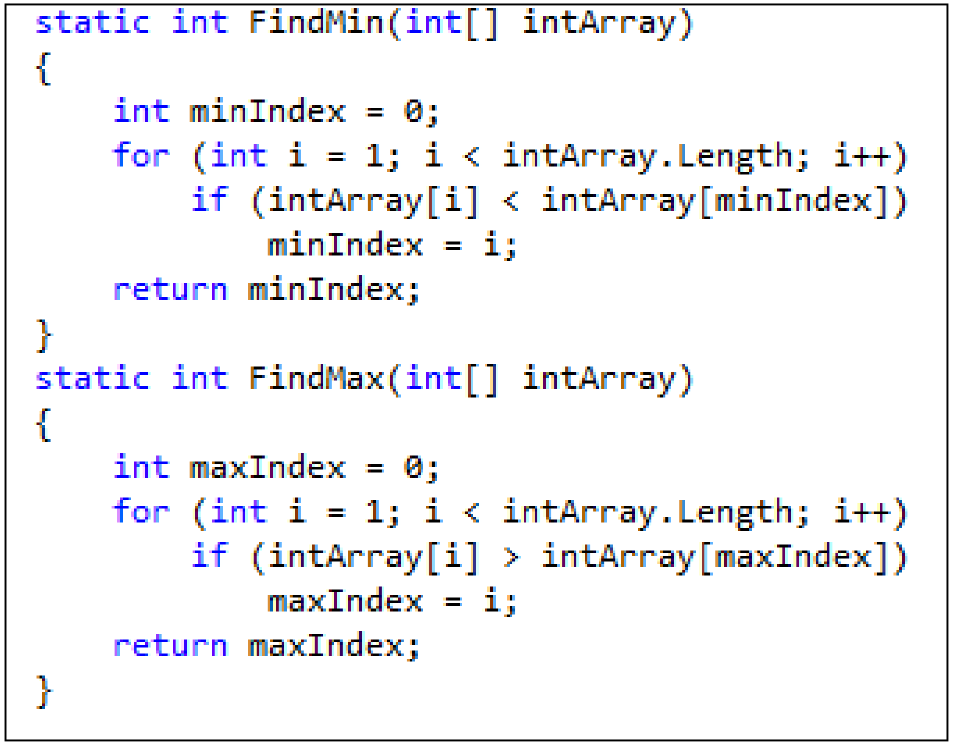
- Inspect the SequentialSearch function
- In case there are many copies of the searched value in the array, the index of which value will be returned?
- What is the Big O complexity of the sequential search algorithm
- What do think if the searched array is sorted



```
namespace BasicSearching
{
    class Example1
    {
        static void Main()
        {
            // ceate and fill the array
            int[] nums = new int[10];
            Random rnd = new Random();
            for (int i = 0; i < 10; i++)
                nums[i] = rnd.Next(100);
            DisplayIntArray(nums);
            // search a value
            int value=0;
            while (true)
            {
                Console.WriteLine("Enter integer value to search for, or -2 to exit:");
                if (value == -2) break;
                value = int.Parse(Console.ReadLine());
                if (SequentialSearch(nums, value) != -1)
                    Console.WriteLine("{0} exists in the array", value);
                else
                    Console.WriteLine("{0} does not exist in the array", value);
            }
            Console.WriteLine("Press any key to continue:");
            Console.Read();
        }
        static int SequentialSearch(int[] intArray, int searchedValue)
        {
            for (int index = 0; index < intArray.Length; index++)
                if (intArray[index] == searchedValue) return index;
            return -1;
        }
        static void DisplayIntArray(int[] intArray)
        {
            for (int i = 0; i < intArray.Length; i++)
            {
                Console.WriteLine("Element {0} is {1}", i, intArray[i]);
            }
        }
    }
}
```

Sequential search: Searching for Minimum and Maximum Values

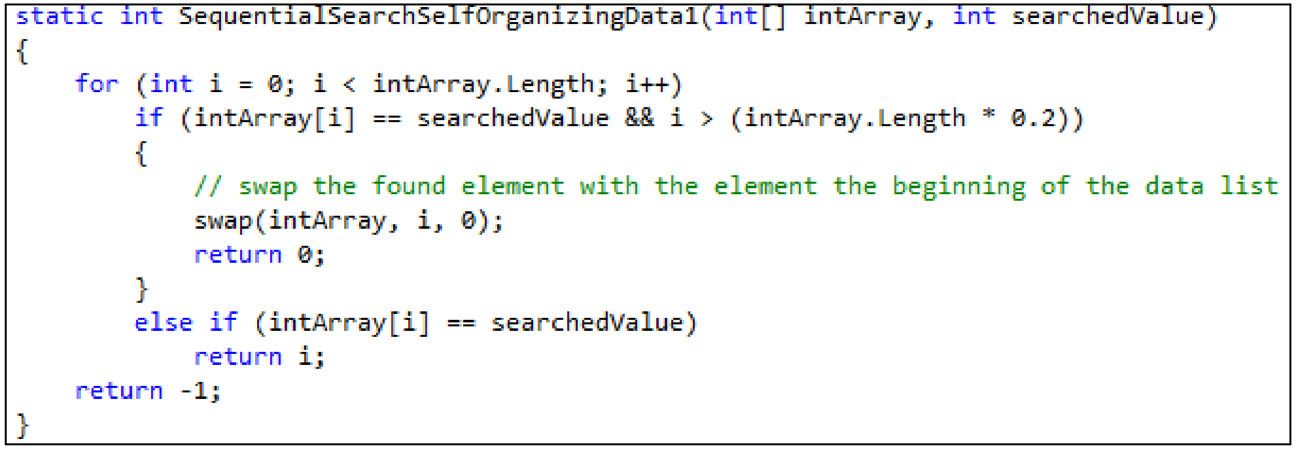
1. Assign the first element index to a variable as the index of the minimum value.
2. Begin looping through the array, comparing each successive array element with the array value pointed by the minimum value index variable.
3. If the currently accessed array element is less than the currently pointed minimum value, assign the index of this element to the minimum value index variable.
4. Continue until the last array element is accessed.
5. The minimum value index is stored in the variable.
6. The process for finding the maximum is similar



```
static int FindMin(int[] intArray)
{
    int minIndex = 0;
    for (int i = 1; i < intArray.Length; i++)
        if (intArray[i] < intArray[minIndex])
            minIndex = i;
    return minIndex;
}
static int FindMax(int[] intArray)
{
    int maxIndex = 0;
    for (int i = 1; i < intArray.Length; i++)
        if (intArray[i] > intArray[maxIndex])
            maxIndex = i;
    return maxIndex;
}
```

Making Sequential Search Faster: Self-Organizing Data

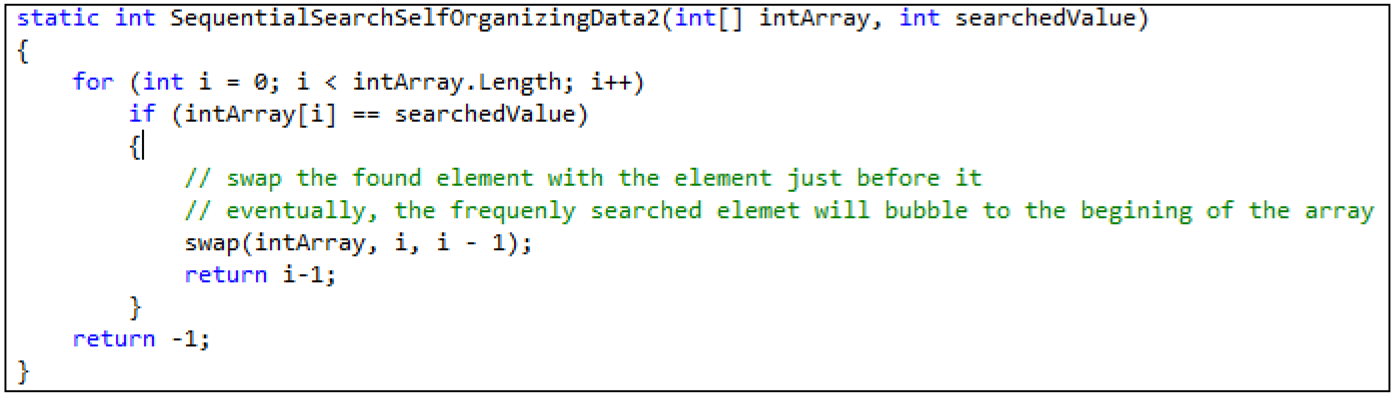
- The 80%-20% rule: it’s often that 80% of data searching operation search for only 20% of the data.
- If this is the case a self organizing searching improve performance
- One way to take this into account is to swap the found element with one of the first 20% elements if it’s not already there



```
static int SequentialSearchSelfOrganizingData1(int[] intArray, int searchedValue)
{
    for (int i = 0; i < intArray.Length; i++)
        if (intArray[i] == searchedValue && i > (intArray.Length * 0.2))
        {
            // swap the found element with the element the beginning of the data list
            swap(intArray, i, 0);
            return 0;
        }
        else if (intArray[i] == searchedValue)
            return i;
    return -1;
}
```

Making Sequential Search Faster: Self-Organizing Data

- The 80%-20% rule: it’s often that 80% of data searching operation search for only 20% of the data.
- If this is the case a self organizing searching improve performance
- Another way is to swap the found element with the element just before it. In this way the frequently search element will bubble up to the beginning of the data list



```
static int SequentialSearchSelfOrganizingData2(int[] intArray, int searchedValue)
{
    for (int i = 0; i < intArray.Length; i++)
        if (intArray[i] == searchedValue)
        {
            // swap the found element with the element just before it
            // eventually, the frequenly searched elemet will bubble to the begining of the array
            swap(intArray, i, i - 1);
            return i-1;
        }
    return -1;
}
```

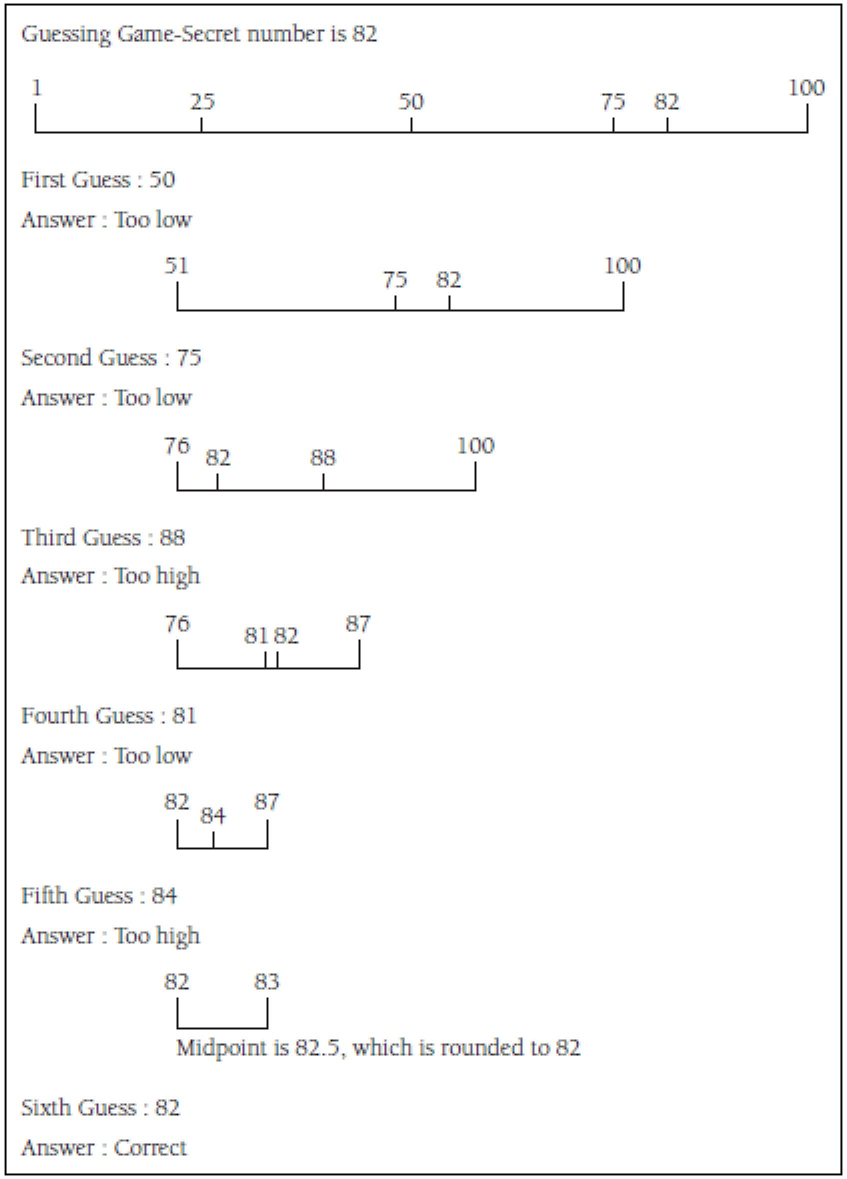
Report Discussion

Last lecture report: : Time comparison between the three basic sorting algorithms on different array sizes

New report: Time comparison between iterative and recursive implementation of the binary search algorithm

Binary search

- We first need our data stored in an array
- The first steps in the algorithm are to set the lower and upper bounds of the search. At the beginning of the search, they are the lower and upper bounds of the array
- Then, we calculate the midpoint of the array by adding the lower and upper bounds together and dividing by 2. The array element stored at this position is compared to the searched-for value
- If they are the same, the value has been found and the algorithm stops.
- If the searched-for value is less than the midpoint value, a new upper bound is calculated by subtracting 1 from the midpoint.
- Otherwise, if the searched-for value is greater than the midpoint value, a new lower bound is calculated by adding 1 to the midpoint
- The algorithm iterates until the lower bound equals the upper bound, which indicates the array has been completely searched and the value not found



Guessing Game-Secret number is 82

First Guess : 50
Answer : Too low

Second Guess : 75
Answer : Too low

Third Guess : 88
Answer : Too high

Fourth Guess : 81
Answer : Too low

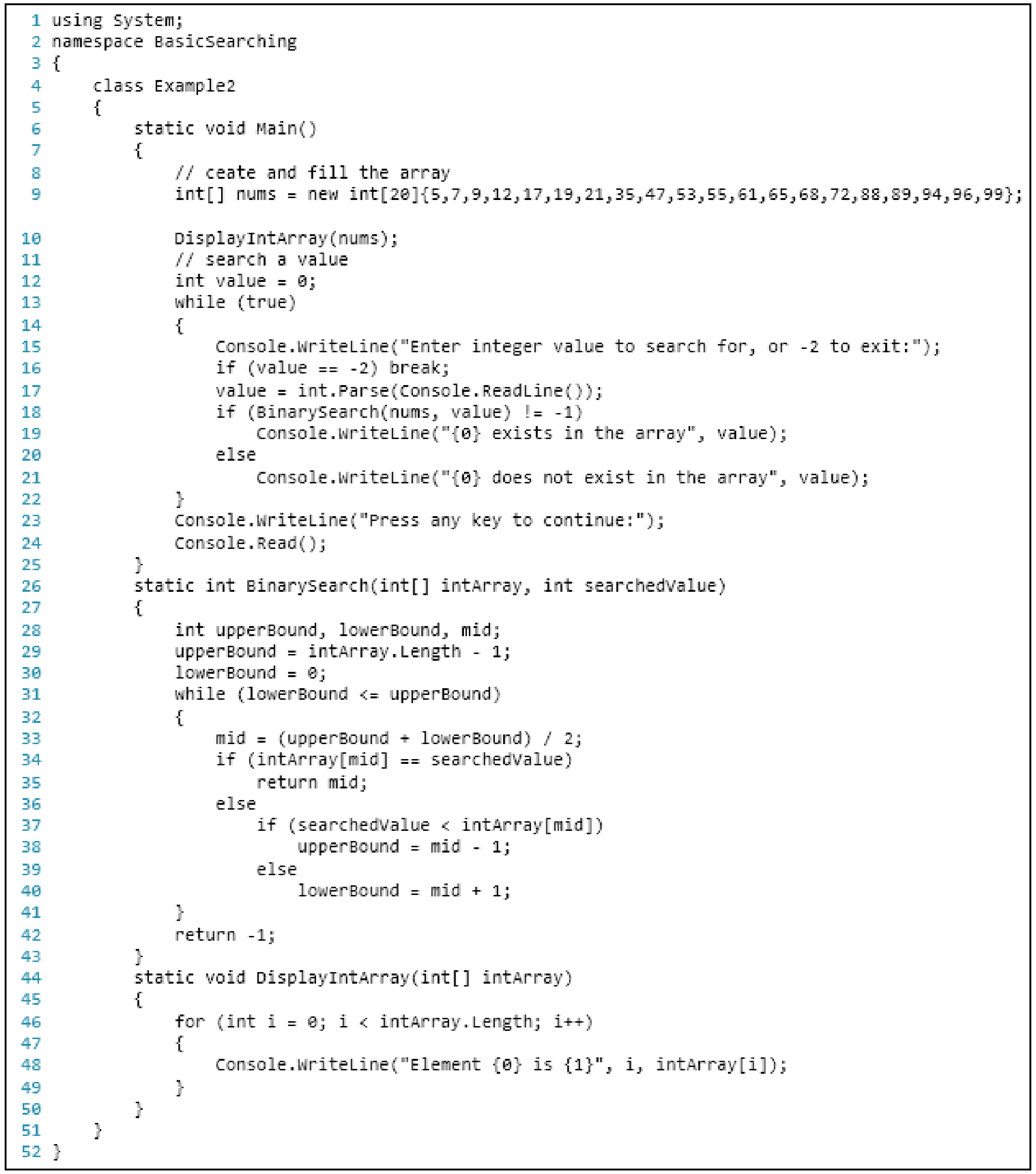
Fifth Guess : 84
Answer : Too high

Midpoint is 82.5, which is rounded to 82

Sixth Guess : 82
Answer : Correct

Iterative Binary search implementation

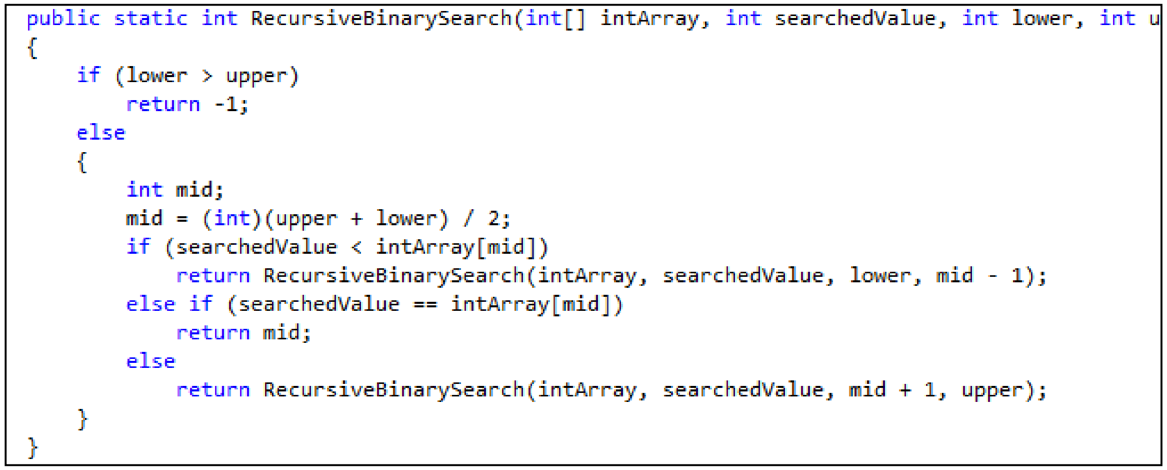
- Why this approach is called iterative?
- Is this algorithms assumes ascending or descending sort order? How to modify it to the other?
- What happens if midd yields fractional number?



```
using System;
namespace BasicSearching
{
    class Example2
    {
        static void Main()
        {
            // ceate and fill the array
            int[] nums = new int[20]{5,7,9,12,17,19,21,35,47,53,55,61,65,68,72,88,89,94,96,99};
            DisplayIntArray(nums);
            // search a value
            int value = 0;
            while (true)
            {
                Console.WriteLine("Enter integer value to search for, or -2 to exit:");
                if (value == -2) break;
                value = int.Parse(Console.ReadLine());
                if (BinarySearch(nums, value) != -1)
                    Console.WriteLine("{0} exists in the array", value);
                else
                    Console.WriteLine("{0} does not exist in the array", value);
            }
            Console.WriteLine("Press any key to continue:");
            Console.Read();
        }
        static int BinarySearch(int[] intArray, int searchedValue)
        {
            int upperBound, lowerBound, mid;
            upperBound = intArray.Length - 1;
            lowerBound = 0;
            while (lowerBound <= upperBound)
            {
                mid = (upperBound + lowerBound) / 2;
                if (intArray[mid] == searchedValue)
                    return mid;
                else
                    if (searchedValue < intArray[mid])
                        upperBound = mid - 1;
                    else
                        lowerBound = mid + 1;
            }
            return -1;
        }
        static void DisplayIntArray(int[] intArray)
        {
            for (int i = 0; i < intArray.Length; i++)
            {
                Console.WriteLine("Element {0} is {1}", i, intArray[i]);
            }
        }
    }
}
```

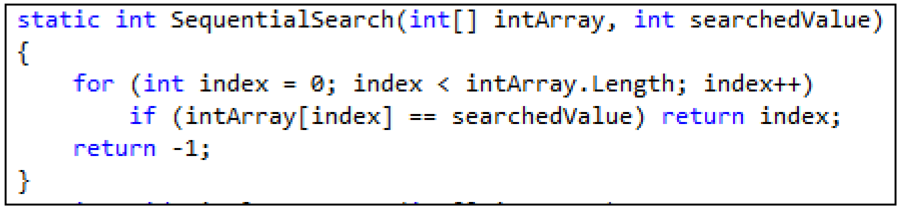
Recursive binary search implementation

- Binary search is a reclusive operation : The binary search algorithm is really a recursive algorithm because, by constantly subdividing the array until we find the item we're looking for (or run out of room in the array), each subdivision is expressing the problem as a smaller version of the original problem.
- Which is better from the performance point of view: Iterative or recursive implementation? Iterative is far better because of the cost of function calls but recursive design is more natural and readable and in some cases it's the only way to go.

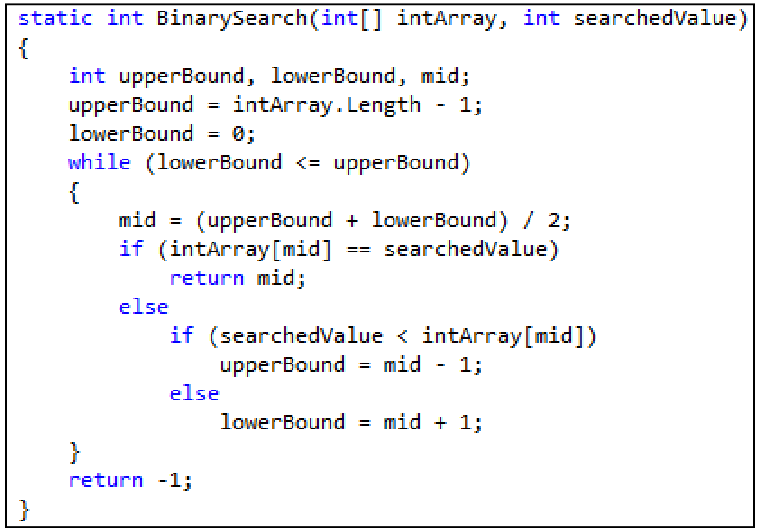


```
public static int RecursiveBinarySearch(int[] intArray, int searchedValue, int lower, int u
{
    if (lower > upper)
        return -1;
    else
    {
        int mid;
        mid = (int)(upper + lower) / 2;
        if (searchedValue < intArray[mid])
            return RecursiveBinarySearch(intArray, searchedValue, lower, mid - 1);
        else if (searchedValue == intArray[mid])
            return mid;
        else
            return RecursiveBinarySearch(intArray, searchedValue, mid + 1, upper);
    }
}
```

Complexity analysis



```
static int SequentialSearch(int[] intArray, int searchedValue)
{
    for (int index = 0; index < intArray.Length; index++)
        if (intArray[index] == searchedValue) return index;
    return -1;
}
```



```
static int BinarySearch(int[] intArray, int searchedValue)
{
    int upperBound, lowerBound, mid;
    upperBound = intArray.Length - 1;
    lowerBound = 0;
    while (lowerBound <= upperBound)
    {
        mid = (upperBound + lowerBound) / 2;
        if (intArray[mid] == searchedValue)
            return mid;
        else
            if (searchedValue < intArray[mid])
                upperBound = mid - 1;
            else
                lowerBound = mid + 1;
    }
    return -1;
}
```

Clearly if all orders are equally probable, the sequential search executes the comparison operation for one time in the best case and for n times in the worst case. In average, the statement executes (n+1)/2 which is O(n)
For binary search, in the worst case the comparison executes; 2 times for 4 items, 3 times for 8 items, 4 times for 16 items, 5 times for 32 items,… in general $LOG_2 n$ times for n items which is O(LOG n)